SELF-MAP: Stochastic Evolution LUT-FPGA Technology Mapper

by

Ahmad Saleh Abdallah Al-Mulhem

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

July, 1996

SELF-MAP : Stochastic Evolution LUT-FPGA Technology Mapper

BY

Ahmad Saleh Abdallah Al-Mulhem

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
Computer Engineering

July 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

Ahmad Saleh Abdallah Al-Mulhem

under the direction of his thesis advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

2.7.96
Dr. Alaaeldin Amin (Chairman)

Dr. Habib Youssef (Co-Chairman)

2/7/96
Dr. Muhammad Al-Suwaiyel (Member)

Department Chairman

Dean, College of Graduate Studies

3.7.96
Date

Dedicated to

my parents

Acknowledgments

First and foremost, all praise to the Almighty Allah who gave me the courage and patience to carry out this work.

Acknowledgment is due to King Fahd University of Petroleum and Minerals for providing support to this work.

My deep appreciations go to my thesis committee, Dr. Alaaeldin Amin, Chairman, Dr. Habib Youssef, Co-Chairman, and Dr. Muhammad Al-Suwaiyel, member.

Thanks are also due to my brother Dr. Muhammed Al-Mulhem for the valuable advices and unconditional continuous support.

Finally, my profound gratitude and appreciation go to the rosy part of my life, to my parents and dear sisters and brothers for their continuous prayers, encouragement and moral support.

Contents

List of Tables

List of Figures

Abstract

Name: Ahmad Saleh Abdallah Al-Mulhem

Title: SELF-MAP : Stochastic Evolution LUT-FPGA Technology Mapper

Major Field: Computer Engineering

Date of Degree: July, 1996

The Technology Mapping Problem for *Look-Up table* based *Field Programmable Gate Arrays* (LUT-FPGAs) is a very important part of a typical CAD system for LUT-FPGAs. The problem has recently been the subject of extensive research. With the exception of a Genetic Algorithm approach, all solutions reported in the literature have used *constructive heuristics.* In this thesis, a new *iterative* technology mapper for Look-Up Table based Field Programmable Gate Arrays (LUT-FPGA) has been designed and implemented. The new technology mapper (SELF-Map) is based on the Stochastic Evolution Algorithm and can optimize circuits for area, delay, or any area-delay combination. It is capable of mapping any arbitrary combinational Boolean network to any LUT-Based FPGA provided that the LUTs have a single output and K inputs where K is an integer number. SELF-Map has been implemented and tested on several benchmark circuits and was shown to generally perform better than other algorithms reported in the literature.

Master of Science Degree

King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

July, 1996

بسم الله الرحمن الرحيم

خلاصة الرسالة

الإسم : أحمد صالح عبدالله ألملحم

عنوان الرسالة : SELF-MAP : محول تقني لمصفوفات البوابات القابلة للبرمجة والتي تعتمد على ذاكرات البحث (LUT-FPGA) مبني على طريقة التطور العشوائي (Stochastic Evolution Algorithm)

التخصص : هندسة الحاسب الآلي

تاريخ الشهادة : صفر ١٤١٧ هـ

يعتبر التحويل التقني (technology mapping) خطوة مهمة في أي نظام للتصميم بواسطة الحاسب لمصفوفات البوابات القابلة للبرمجة والتي تعتمد على ذاكرات البحث (LUT-FPGAs). فقد حظيت هذه المسألة باهتمام الكثير من الباحثين ، و تعتمد جميع الحلول المقترحة على طرق بنائية (constructive techniques) ماعدا واحدة بنيت باستخدام الطريقة الجينية (Genetic Algorithm). أما في هذه الأطروحة فقد تم تصميم و تنفيذ طريقة جديدة (SELF-Map) مبنية على طريقة التطور العشوائي (Stochastic Evolution Algorithm). وقد تم اختبار هذه الطريقة على عدد من الدارات الأختبارية (benchmark circuits) و أثبتت تفوقا على الطرق الموجودة من حيث ايجاد حلول ذات كفائة عالية في مساحة الحل المقترح و سرعته.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

صفر ١٤١٧ هـ

Chapter 1

FIELD PROGRAMMABLE GATE ARRAYS

1.1 Introduction

VLSI technology has made it possible to put millions of transistors on a single chip. It allows implementing a wide range of *Application Specific Integrated Circuits* (ASICs), using either *full custom approach*, or *semi-custom approach*. In the full custom approach, all parts of the circuit are carefully designed to meet specific requirements. Therefore, this approach gives the best design for a given circuit. However, it takes a lot of time for the design to be completed, and it is very expensive unless the circuit will be produced in large volumes. Building *microprocessors* is an example where using full custom approach is justified. However, for lower volume

circuits, such as ASICs, the semi-custom approach is the natural choice.

Under the semi-custom approach, there are several design styles. The following are the most widely used semi-custom design styles:

1. standard cells,

2. gate arrays,

3. Field Programmable Gate Arrays (FPGAs).

These design styles differ in their performance, silicon area utilization, *turn-around-time* (TAT), etc.

The development of *Field Programmable Gate Arrays* (FPGAs) represents an important evolutionary step in the area of programmable logic devices. FPGAs are fairly versatile and flexible VLSI chips that are desk-top user-programmable, which allow the realization of a wide range of designs in very short periods of time. Some FPGAs are re-programmable which makes them ideal for fast *prototyping* applications. The first FPGA was introduced in 1985 by Xilinx Corp [28]. Soon, other manufacturers introduced other types of FPGAs.

The architecture of an FPGA varies from one manufacturer to another [2]. However, an FPGA typically consists of three main parts (Figure 1.1):

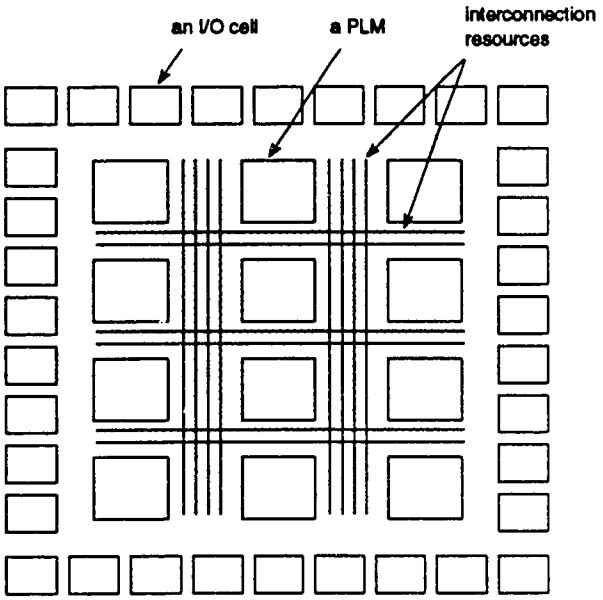


Figure 1.1: The structure of a typical FPGA.

1. a 2-D array of programmable logic modules (PLMs),

2. programmable interconnection resources, and

3. programmable I/O cells.

The *programmable logic module* (PLM) can be as simple as a 2-input NAND gate, or as complex as a *K-input RAM Look Up Table* (K-LUT) with Flip-Flops. The *programmable interconnection resources* include horizontal and vertical *wire segments*, and *programmable switches*. These are programmed so as to realize the required interconnections between the PLMs. The *programmable I/O cell* is a buffer which can be programmed as an input, output, or input/output buffer to interface

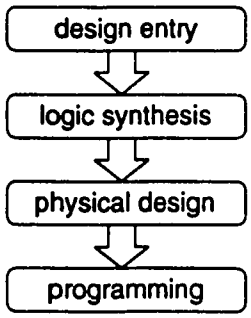


Figure 1.2: The design steps for an FPGA.

with the outside world.

1.2 CAD System For FPGAs

An efficient CAD system is a basic requirement for any ASIC design style. FPGAs are no exception. Figure 1.2 shows a typical CAD system for mapping a given design into FPGA implementation. It consists of the following four main steps:

1. design entry,
2. logic synthesis,
3. physical design, and
4. programming.

In the *design entry*, the design is described using a hardware description language (e.g., VHDL), as a schematic drawing, or as a set of Boolean expressions.

The design entry step usually transforms the design into some intermediate form that is suitable for the succeeding steps.

The *logic synthesis* step usually consists of two main phases [27].

1. A technology-independent optimization phase, and

2. A technology mapping phase.

The *technology-independent optimization phase* uses a collection of general optimization routines that are suitable for any technology. Typically, they manipulate the design to improve its performance, or reduce its area requirement. For instance, to reduce area, the SIS package [27] minimizes the number of literals by removing redundancy and extracting common sub-expressions. The *technology mapping phase* receives the optimized design and implements it on an FPGA platform. The objective of this phase can be the minimization of the number of PLMs (area minimization), the minimization of the PLMs' depths (delay minimization), or a combination of both. Other objectives have been also considered, such as routing [26].

The *physical design* step usually consists of two main phases[24].

1. placement, and

2. routing.

The *placement* phase assigns every PLM generated by the technology mapping phase to a physical location in the FPGA's array. The assignment is usually done so as to minimize the total wire length required to connect the placed PLMs. The *routing* phase establishes the required connections between PLMs by selecting wire segments and programmable switches. The objective of the routing step is to achieve full connectivity of the design.

In the final step (*programming*), the different programmable elements of the FPGA chip are configured to realize the required design. It is worth mentioning that the whole process, excluding the design entry step, typically takes less than an hour.

1.3 Thesis Outline

In this work, we are concerned with the problem of Technology Mapping in the logic synthesis step. In chapter 2, we introduce some related definitions, and formally state the technology mapping problem. In chapter 3, most of the reported work in the literature is reviewed. In chapter 4, a new technology mapper is introduced and discussed. Some implementation issues and experimental results are presented in chapter 5. Chapter 6 contains a summary of accomplished work and provides directions for further research.

Chapter 2

THE TECHNOLOGY MAPPING PROBLEM

2.1 Introduction

A *Look-up table based FPGA* (LUT-FPGA) is a class of FPGAs where the PLMs are based on RAM Look-up tables (LUT). A K-input LUT (K-LUT) can implement any Boolean function with at most K inputs. In other words, a K-LUT can implement any one of the possible 2^{2^k} Boolean functions.

The technology mapping problem for LUT-FPGAs has been receiving increasing attention due to its importance in the FPGA design flow as outlined in section 1.2. In the literature, this problem has been formulated as a graph problem [6]. In

this chapter, some related definitions are introduced, and the technology mapping problem is formally stated.

2.2 Definitions

A combinational logic circuit can be represented as a *directed acyclic graph* (DAG) $G(V, E)$ where each node v in V represents a Boolean function and each directed edge (u, v) represents a connection between the output of u and the input of v. This DAG representation is usually called a *Boolean Network* (BN). A *Primary input* (PI) is a node with no in-coming edges. Similarly, a *Primary output* (PO) is a node with no out-going edges.

For a node $v \in V$, *input(v)* is the set of nodes that supply inputs to v. In general, given a subset V_1 of V, $input(V_1)$ is the set of nodes in $V - V_1$ that supply inputs to nodes in V_1. For a set S of elements, $| S |$ is the number of elements in S. A node u is a *predecessor* of node v if there is a directed path from u to v in the BN.

Definition 2.1 *A K-feasible cone at a node v, denoted by C_v, is defined as a sub-graph consisting of v and its predecessors such that any path connecting a node in C_v and v lies entirely in C_v, and $| input(C_v) | \leq K$ (see Figure 2.2).*

In this work, we assume that each K-LUT is a programmable block with K inputs

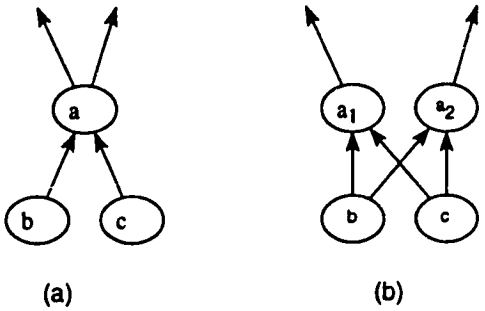


Figure 2.1: The effect of replicating a fanout node.

and one output. Therefore, a K-LUT can implement (or cover) any K-feasible cone in a BN.

Definition 2.2 *The depth of a node v is defined as the maximum number of LUTs (cones) along any path from any primary input to v.*

The depth of a primary input is zero. The depth of a BN is the largest node depth in the BN.

A node v with one out-going edge is termed a *fanout-free* node. A node v with two or more out-going edges is termed a *fanout* node. A fanout node can be replaced with two or more fanout-free nodes without affecting the BN functionality. This can be done by *replicating* a fanout node as shown in Figure 2.1. In Figure 2.1.a, node a is a fanout node. In Figure 2.1.b, node a has been replicated, i.e. replaced with two fanout-free nodes, a_1 and a_2, which have the same function of the original node a.

Definition 2.3 *A potential fanout node is a fanout-free node which feeds a fanout node.*

If a fanout node is replicated then its immediate predecessors become fanout nodes. In Figure 2.1, node a is a fanout node while nodes b and c are potential fanout nodes. As shown in Figure 2.1.b, if node a is replicated (a_1 and a_2), then nodes b and c become fanout nodes.

2.3 Problem Statement

A technology mapper for a K-LUT FPGA has the following features.

- **The input** is a combinational Boolean network the nodes of which represent simple gates, i.e. OR, AND, and NOT, etc.
- **The output** is an equivalent Boolean network the nodes of which represent FPGA K-LUTs,
- **The objective** can be
  - the minimization of the number of LUTs (area minimization) [7, 8, 15, 6, 18, 12],

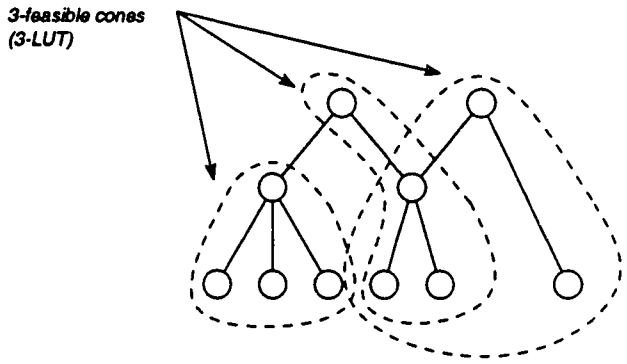


Figure 2.2: An example of a Boolean network and its mapping.

- the minimization of the LUTs' depth (delay minimization) [9, 3, 4],
- combined area-delay minimization [5], or
- easing routability [26].

For $K \geq 5$, this problem is NP-complete [6]. Figure 2.2 shows an example of a Boolean network and a possible mapping with 3-feasible cones; or equivalently with 3-LUTs.

Chapter 3

PREVIOUS WORK

3.1 Introduction

In the literature, many heuristic algorithms have been proposed to solve the technology mapping problem for LUT-FPGAs. In this chapter, we review most of the reported heuristics.

3.2 DAG-Map

The DAG-Map [3] package consists of three major parts; namely

1. **Network Transformation:** A preprocessing routine transforms an arbitrary Boolean network into a two-input network, i.e. a network with all nodes having at most two inputs.

2. **Technology Mapping:** The DAG-Map algorithm maps the two-input network into a K-LUT FPGA network with minimum delay.

3. **Area Minimization:** A postprocessing performs area optimization of the FPGA network without increasing network delay.

3.2.1 Network Transformation

The transformation is done for two reasons. First, it avoids the need for node decomposition in the technology mapping part. Second, the authors assume that smaller objects can be more efficiently packed.

They developed an algorithm (DMIG) to perform such transformation. The algorithm traverses the network starting at primary inputs proceeding towards the primary outputs. As nodes are visited, it decomposes any node with more than two inputs into a tree of two-input nodes. They proved that DMIG increases the network depth by a small constant factor.

3.2.2 Technology Mapping

The technology mapping algorithm is based on *Lawler's labeling algorithm* [16] which consists of two phases. The first phase assigns a label to every node, which is equal to the level of the K-LUT containing the node in the final mapping solution. Clearly,

it is desirable to have the label of a node to be as small as possible. To do that, the nodes are traversed in topological order starting at the primary inputs. Then, for every node, the algorithm checks whether it can be clustered with its predecessors without increasing the number of inputs to the cluster beyond k. If that is possible, then the node is given a label equal to the label of its predecessor. Otherwise, the label of the node is equal to the label of its predecessors plus one.

The second phase generates the K-LUTs in the mapping solution. In this phase, the network is traversed starting at the primary outputs assigning a K-LUT to every cluster generated in the first phase.

3.2.3 Area Optimization

DAG-Map has a secondary objective which is minimizing the area, without increasing the depth of the solution. Two postprocessing operations are used to accomplish that. The first operation is gate decomposition using Roth-Karp method [20]. The second operation is predecessor packing where each node is checked if it can be packed with one or more of its single fanout predecessors.

3.3 FlowMap

Unlike other mappers, FlowMap [4] optimally solves the LUT-based FPGA technology mapping problem for a general Boolean network to minimize the delay. Other mappers are guaranteed to be optimal if the network is fanout-free. Some mappers even add more constraints. Chortle-crf [8], for example, requires that the LUT has at most 5 inputs.

Using *the unit delay model*, the depth of the mapped solution is taken as a measure for the network delay. Therefore, the objective is to have the minimum network depth. Two factors contribute to the delay; namely the delay in the K-LUTs, which is a constant, and the delay in the interconnections. The delay in the interconnections is approximated by the number of edges.

FlowMap models the mapping problem as a network flow problem [17]. A key factor behind this clever formulation is to compute a minimum height K-feasible cut in a network, which is solved in a polynomial time.

FlowMap applies the same overall approach used in DAG-Map [3], which is as follows:

1. **Network Transformation:** A preprocessing procedure transforms an arbi-

trary Boolean network into a two-input network, i.e. a network with all nodes having two inputs at most. The same DMIG routine is used in this part.

2. **Technology Mapping:** An algorithm maps the two-input network into a K-LUT FPGA network with minimum delay.

3. **Area Minimization:** A postprocessing routine performs area optimization of the FPGA network without increasing the network depth.

3.3.1 Technology Mapping

As in DAG-Map [3], this part is based on *Lawler's Labeling Algorithm* [16], which consists of two phases. The first phase is to label all the nodes in a Boolean network N. The labeling is done in topological order starting at the primary inputs. For each PI node $u, l(u) = 0$, where $l(u)$ is the label of node u.

The process of labeling a node with the minimum label has been formulated as finding *the minimum height K-feasible cut* [10]. An important contribution of this work is an $O(Km)$ algorithm to find the minimum K-feasible cut, where m is the number of edges.

The second phase generates the K-LUTs in the mapping solution. This phase is the same as in DAG-Map. The network is traversed starting at the primary outputs

assigning a K-LUT to every cluster generated in the first phase.

3.3.2 Area Minimization :

As a secondary objective, FlowMap minimizes the area without increasing the depth of the mapping solution. Two postprocessing are used:

1. **Maximizing the cut volume during mapping:** In general, the minimum height K-feasible cut is not unique. Intuitively, the larger the cut volume, the more nodes we can pack into a K-LUT.

2. **Flow pack:** This is a generalized version of the predecessor packing algorithm used in DAG-Map. In the predecessor packing algorithm, a node is checked if it can be packed with one of its predecessor. In flow pack, however, a node is checked if it can be packed with a set of its predecessors.

3.3.3 Area/Depth Trade-Off :

In [5], the authors studied the area and depth trade-off in LUT-FPGA technology mapping. Starting with a depth-optimal mapping which is obtained using FlowMap [4], a sequence of depth relaxation operation and area minimization procedures are performed to produce a set of mapping solutions with smooth area-depth trade-off.

3.4 The Chortle Algorithms

Chortle [7] and its extensions Chortle-crf [8], and Chortle-d [9] accept an AND/OR representation of a Boolean network. Inversions are represented by labels on the edges. The network is first decomposed into a forest of fanout free trees, by clipping the multiple fanout nodes. The minimum cost mapping of each tree into LUTs is then performed using dynamic programming. The resulting LUTs are then assembled according to the interconnection patterns of the forest.

In the process of finding the optimal mapping of a tree (dynamic programming), the tree is traversed beginning at the primary inputs and proceeding towards the primary output. In the case of Chortle and Chortle-crf, the objective is to minimize the number of LUTs. On the other hand, Chortle-d has the objective of improving the circuit performance by minimizing the depth (delay).

3.4.1 Chortle

Finding the minimum number of LUTs implementing a node is done by exhaustively searching all the possibilities of combining the node with its fanin. Also, if the number of inputs to the node is more than k, then another exhaustive search of all possible decompositions is done. The exception is that, if k is larger than 10, then the possible decompositions become very large. Therefore, the node is first split into

two nodes with roughly equal fanin, then the two nodes are decomposed separately. At this point, optimality is not guaranteed any more.

3.4.2 Chortle-crf

The major contribution of Chortle-crf [8] is the use of *bin packing* to choose gate-level decompositions. Moreover, Chortle-crf considers two important aspects which results in further reduction in the number of LUTs in the circuit; namely the exploitation of reconvergent paths and replication of logic at fanout nodes. These features are explained in the following sections.

3.4.3 Bin Packing Approach

The key to constructing the best circuit implementing a node is finding a node decomposition, that reduces the number of LUTs in the final circuit. The best decomposition is a tree of LUTs that implements both the function of the fanin LUTs and a decomposition of the node. The tree is constructed in two steps. First, a two-level decomposition is constructed then this decomposition is converted into a multi-level decomposition.

The result of a two-level decomposition is a single first-level node and several second-level nodes. The decomposition is constructed using a *FirstFit Decreasing* bin packing algorithm. The bins are the second-level LUTs, which have the capacity

of k, and the boxes are the fanin nodes.

The decomposition is completed by implementing the first-level node with a tree of LUTs. Basically, any second-level LUT with unused inputs can be used to implement a portion of the first-level node. This final multi-level decomposition can be shown to be optimal if the network is a fanout free tree and the value of k is less than 6.

It is worth mentioning that the application of the bin packing algorithm made Chortle-crf [8] up to 28 times faster than the exhaustive approach used in Chortle [7].

3.4.4 Exploiting Reconvergent Paths

Chortle-crf looks for reconvergent paths, which exist if a pair of fanin nodes share some inputs. In general, if two boxes share some inputs, and the total number of distinct inputs to these two boxes is less than or equal to k, then it is possible to pack the two boxes in one bin.

The process of merging pairs of boxes may interfere with the bin packing algorithm which is applied next. So, Chortle-crf considers both the packing with merging allowed and the packing with merging disallowed. The best result is then selected.

3.4.5 Replication of Logic at Fanout Nodes

If a fan-out node is encountered in traversing a network, Chortle-crf considers two options. The first option is that, the node could be implemented explicitly as an output of a LUT. So later, it is treated as a primary input. The other option is to implement it implicitly; i.e a replica of the function is made at every fanout edge.

3.4.6 Chortle-d

Chortle-d [9] applies the same overall approach proposed in Chortle-crf. However, a different procedure is used to construct the decomposition tree for nodes in a network. The decomposition is to minimize the depth.

The first step in constructing the decomposition tree is to separate the fanin lookup tables into strata, where each stratum contains all LUTs at the same depth. Then, the number of LUTs in each stratum is minimized using the same FirstFit Decreasing bin packing algorithm applied in Chortle-crf.

The decomposition tree is completed by proceeding from the uppermost stratum to the deepest stratum, connecting the outputs of lookup tables in stratum D to unused inputs of lookup tables in stratum $D + 1$. This may require the addition of extra lookup tables in stratum $D + 1$. The decomposition is complete when a

stratum containing only one lookup table is reached, with no other deeper strata remaining.

Similar to Chortle-crf, Chortle-d is optimal if the network is a fanout-free network and $K < 6$.

3.4.7 Mapping for Xilinx

Taking advantage of Xilinx, Chortle looks for pairs of LUTs that can be merged into a single CLB. Finding the maximum number of such pairs is restated as a Maximum Cardinality Matching problem [10].

3.5 Mis-pga

Mis-pga [18] adopts a two step approach. First, the network is transformed into a feasible network, i.e. a network where all nodes have a maximum fanin of K. Then, the number of nodes is reduced by solving a covering problem.

For the first step, the transformation is achieved by decomposing all nodes with fanin more than k. There are several decomposition methods. Mis-pga, uses two decomposition methods:

1. Roth-Karp Decomposition [20].

2. Partition: It is based on the notion of kernel extraction [25]. For every non-feasible function, it extracts all possible kernels. Then, it chooses the kernel with the least cost, where for every kernel k_i, and residue r_i that are feasible functions, the cost is calculated as follows:

$$cost(k_i) = \mid sup^1(k_i) \cap sup(r_i) \mid$$

3.5.1 Covering

First, all supernodes corresponding to each node of the network are generated by repeatedly applying the maxflow algorithm. A supernode corresponding to a node i of a boolean network is defined as a cluster that contains the node i, and some nodes in the transitive fanin of the node i. A cluster can not contain any primary inputs. Every node in the cluster has a path to the node i which lies completely within the cluster. A supernode must have a maximum of K inputs.

After finding all the possible supernodes, the covering solution is found by selecting a set of supernodes which cover the network, such that every intermediate node in the network exist in at least one supernode. Also, if a supernode is chosen, then for each input to the supernode, some supernodes whose output supply the input must also be chosen.

[1] *sup(f)* stands for the set of variables that a node *f* explicitly depends on.

This covering problem has been formulated as a *binate covering problem*, which is described in [21].

3.5.2 Mapping for Xilinx

Taking advantage of Xilinx, mis-pga looks for pairs of LUTs that can be merged into a single CLB. Finding the maximum number of such pairs is restated as a Maximum Cardinality Matching problem [10].

3.6 Xmap

At the top level, Xmap [12] is a two-steps mapper which

1. transforms the input network into a feasible one by decomposing non-feasible nodes.

2. covers the resulting transformation by LUTs.

Although, the same is done in other mappers, the details of transformation and covering are different here.

To achieve the first step, Xmap converts the input network into an *if-then-else* DAG representation. This conversion results in a Boolean network with all nodes having three inputs.

For the covering step, Xmap traverses the if-then-else DAG from the primary inputs to the primary outputs and keeps a record of the number of inputs that are seen in the path connecting the primary inputs to the node under consideration. If the node has more than K inputs some of its predecessors must be placed in a different LUT. A heuristic routine is used to select the predecessor. The selection is based on the number of the predecessors inputs. The more inputs they can isolate from the node under consideration the better.

As a final step, Xmap takes advantage of Xilinx CLB architecture which allows two LUTs to be implemented in one CLB, provided that their total number of distinct inputs is less than or equal to 5, and they share 3 inputs. A greedy algorithm is applied which takes as input the following:

L: the set of logic blocks (LUT).

i: the maximum number of inputs for each function in a shared CLB.

t: the maximum total number of inputs in shared CLB.

The algorithm is as follows :

1. All blocks with more than i inputs are removed from L. They are not mergeable.

2. The remaining set is sorted by the number of inputs to each block.

3. The block with maximum number of inputs is removed from L, and mergeablity is checked with all remaining blocks, in decreasing number of inputs.

4. If a legal merging is found, then the other block is removed from L.

5. Repeat 3 and 4, until L is empty.

3.7 Level-Map

Farrahi et al. [6] have shown that the technology mapping problem for LUT-FPGAs is NP-complete. They formulated the problem as a graph covering problem with the restriction that nodes are not allowed to decompose, and called it K-RLMP for restricted K-LUT minimization problem. Then, they proved that the 3-satisfiability problem (3-SAT) reduces in polynomial time to the K-RLMP, for any value of $K \geq 5$, where K is the LUT capacity. Since, it is well known that the 3-SAT is an NP-complete problem, then K-RLMP for $K \geq 5$ is also an NP-complete problem.

To solve the K-RLMP problem, they proposed an algorithm (Tree-Map) that optimally maps a *tree* into K-LUTs with a running time of $O(min\{nK, n \log n\})$, where n is the number of nodes in the tree. Based on the Tree-Map algorithm, they proposed a polynomial time heuristic algorithm (Level-Map) for general Boolean networks. In the following, the Tree-Map algorithm is presented , followed by the Level-Map algorithm.

```
ALGORITHM Tree-Map(T(V,E));
   FOR every v in V compute Level(v);
   h = the tree height;
   FOR every PI v, set d_v = 1;
   FOR i = 1 TO h DO
      FOR every node v at level i DO
         compute d_v ;
         IF d_v > K THEN
            a: sort the children of v according to d in decreasing order;
            b: assign a LUT to the child c with maximum d;
            c: delete the child from v's children;
            d: update d_v = d_v - (d_c - 1);
            e: IF d_v > K THEN goto b;
         ENDIF
      ENDFOR
   ENDFOR
END (* of Tree-Map(T(V,E)) *);
```

Figure 3.1: Tree-Map Algorithm.

3.7.1 The Tree-Map Algorithm

Figure 3.1 shows an outline of the algorithm. The algorithm starts by levelizing the tree, where leaf nodes (PI) receive a level of 1. The algorithm makes use of another parameter; the dependency of a node v, d_v, which is recursively defined as follows:

$$d_v = \begin{cases} 1 & \text{if } v \text{ is a } PI \\ \sum_{c \in C} d_c & \text{otherwise, where } C \text{ is the set of } v\text{'s children} \end{cases}$$

The algorithm performs a level-wise scan of all nodes, and computes their dependencies d. If the dependency d_v for some node v exceeds K, then the children of v are sorted according to their dependencies in a decreasing order. A child with

maximum d is picked and assigned a LUT, and the dependency of the parent node v is updated. The process is repeated as necessary until d_v is made less than or equal to K.

3.7.2 The Level-Map Algorithm

The Level-Map algorithm is basically the same as the Tree-Map algorithm with two main differences. First, the level calculation step is replaced with a topological sorting step, which serves the same purpose. Second, the sorting of node children is based on a function that accounts for their fanout as well as their dependencies.

3.8 GAFPGA

All the algorithms reviewed so far are constructive heuristics, i.e. they build a solution step-by-step by traversing a Boolean network and deciding for every node whether to assign a LUT or not. GAFPGA [15], however, is the only reported algorithm that applies an iterative heuristic, in this case the Genetic algorithm [11], to optimize for area.

The Genetic algorithm has been adapted to solve the *circuit segmentation* problem. The circuit segmentation problem is defined as follows. Given a combinational logic circuit and a positive integer k, partition the circuit into (not necessarily dis-

joint) subcircuits, such that the number of inputs to each subcircuit does not exceed k and the number of lines connecting different subcircuits is minimized.

The authors developed genetic operators (crossover and mutation) to search the different possible segmentations for a given circuit. They also considered two decomposition possibilities at every node that has more than two inputs.

GAFPGA has been reported to perform better than other constructive algorithms [15]. However, GAFPGA suffers from excessive computation time and the need to validate every solution generated by the genetic operators.

Chapter 4

SELF-MAP

4.1 Introduction

The intractability of the technology mapping problem for LUT-FPGAs has led to a large number of heuristic solution techniques which targeted good solutions rather than optimal ones. The traditional objectives for the technology mapping problem include area minimization by minimizing the number of LUTs, delay minimization by minimizing the LUTs' depth, or a combination of these. All reported heuristics (except GAFPGA [15]) are constructive and deterministic. It is well known that constructive heuristics, because of their greedy nature, frequently get trapped into local optima. The situation is even worse when the problem has several noisy conflicting objective functions. A noisy objective function is one which has a large number of local optima. A class of heuristics that are suitable for such problems are

iterative non-deterministic algorithms. The most well known among these are,

- simulated annealing [13],
- genetic algorithm [11],
- tabu search [1],
- simulated evolution [14], and
- stochastic evolution [23].

All the above iterative search algorithms have several properties in common:

1. they are general and can be used for almost any combinatorial optimization problem;
2. they have an *up-hill-climbing property*, i.e. they occasionally accept worse solutions to allow the algorithm to escape from local optima;
3. they are suitable for combinatorial optimization problems with noisy objective functions.

As mentioned earlier, GAFPGA suffers from the excessive computation time and the need to validate every solution generated by the genetic operators. It is our belief, however, that the full potential of iterative algorithms as applied to the LUT-FPGA technology mapping problem can be further investigated to come up with better solutions which are more computationally efficient. This chapter

describes a new technology mapper for LUT-FPGA which is based on the stochastic evolution algorithm (SE). The SE algorithm has been successfully applied to solve several combinatorial optimization problems [23]. In the following sections, the SE algorithms will be first reviewed, then its application to the technology mapping problem for LUT-FPGA is presented.

4.2 Stochastic Evolution

The Stochastic Evolution algorithm (SE) has been proposed by Saab et al. [23] as a general iterative technique inspired by the alleged evolution process in biological systems. The algorithm is intended to solve a wide range of combinatorial optimization problems with an attempt to overcome some of the drawbacks of the simulated annealing algorithm [13], namely the need for careful tuning of many control parameters, and excessive computation time.

4.2.1 The State Model

Combinatorial optimization problems are problems that seek a global minimum of some real valued cost function $cost : \Omega \rightarrow R$ defined over a *discrete* set Ω. The set Ω is called the state space and its elements are referred to as states. A state space Ω together with an underlying neighborhood structure (the way one state can be reached from another state) form the solution space.

Combinatorial optimization problems can be modeled in a number of ways. SE adopts the following model.

The state model: *Given a finite set M of movable elements and a finite set L of locations, a state is defined as a function* $S : M \rightarrow L$ *satisfying certain constraints.*

Many of the combinatorial optimization problems can be formulated according to this model. Consider, for example, *the Graph Bisection problem*[23].

Problem : Given a graph $G(V, E)$, it is required to partition the graph into two sub-graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ such that

$$|V_1| = |V_2|, \text{ and}$$

$$|V| = |V_1| + |V_2|$$

objective : The number of edges crossing from G_1 to G_2 or vice versa is to be minimum.

To formulate the problem in terms of the above state model, choose $M = V$, the vertex set, and $L = \{1, 2\}$. Then a state is defined as the *onto function* $S : V \rightarrow \{1, 2\}$. In this case, there is one constraint which can be stated as $|S^{-1}(1)| = |S^{-1}(2)|$, i.e. a state is a partition of the vertex set into two parts of equal cardinalities. Moreover, the cost of a state, $cost(S)$, is the number of edges $(i, j) \in E$ with $S(i) \neq S(j)$.

Algorithm SE;
$S_{Best} = S = S_0$;
$C_{Best} = C_{cur} = COST(S)$;
$p = p_0$;
$\rho = 0$;
REPEAT
  $C_{pre} = C_{cur}$;
  $S = PERTURB(S, p)$; /* perform a search in the neighborhood of s */
  $C_{cur} = COST(S)$;
  $UPDATE(p, C_{pre}, C_{cur})$; /* update p if needed */
  IF $(C_{cur} < C_{Best})$ **THEN**
    $S_{Best} = S$;
    $C_{Best} = C_{cur}$;
    $\rho = \rho - R$;
  ELSE
    $\rho = \rho + 1$;
  ENDIF;
UNTIL $\rho > R$;
RETURN(S_{Best});

Figure 4.1: The SE algorithm.

4.2.2 The Algorithm

The idea of the SE algorithm is to find a suitable location $S(m)$ for each movable element $m \in M$ which eventually leads to a lower cost of the whole state $S \in \Omega$. A general outline of the SE algorithm is as shown in Figure 4.1.

The inputs to the SE algorithm are :

1. some initial state (solution) S_0,

2. an initial value of the control parameter p_0, and

3. a stopping criterion parameter R.

The algorithm starts by initializing S, S_{Best}, p, and ρ. Throughout the algorithm, S holds *the current sate (solution)*, while S_{Best} holds *the best state*. If the algorithm generates *a bad state*, then the control parameter p will be used to draw a random number to decide whether to accept this state or not. The input p_0 is used as the initial value of the parameter p. Finally, the variable ρ is used as a counter and initialized to zero. The algorithm terminates when ρ exceeds R, which is also an input parameter.

After initialization, the **REPEAT** structure takes over the algorithm control **UNTIL** the counter ρ exceeds R. Inside the **REPEAT** body, the cost of the current state is first calculated and stored in C_{pre}. Then, the **PERTURB** function is invoked with the current state S and the control parameter p as input parameters. Figure 4.2 shows an outline of the **PERTURB** function.

The **PERTURB** function scans the set of movable elements M according to some apriori ordering and moves every $m \in M$ to a new location $l \in L$. Recall that the current state S is actually a function $S : M \rightarrow L$. When a move is performed, a new state S' is generated, which is *a unique* function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$ for some movable object $m \in M$. To evaluate the move, the gain function $GAIN(m) = COST(S) - COST(S')$ is calculated. If the calculated gain

```
function PERTURB(S, p);
FOR EACH (m ∈ M) according to some apriori ordering DO
    S' = MOVE(S, m);
    GAIN(m) = COST(S) − COST(S');
    IF (GAIN(m) > RANDINT(−p, 0)) THEN
        S = S';
    ENDIF;
ENDFOR;
S =MAKE_STATE(S);
RETURN(S);
```

Figure 4.2: The PERTURB function.

is greater than some integer r, the move is accepted and S' replaces S as the current state. The integer r is randomly generated in the interval $[-p, 0]$, i.e. $-p \leq r \leq 0$. Since $r \leq 0$, moves with positive gain are always accepted. After scanning all the movable elements $m \in M$, the **MAKE_STATE** routine makes sure that the final state satisfies the state constraints. If the the state constraints are not satisfied then **MAKE_STATE** *reverses* the fewest number of latest moves until the state constraints are satisfied.

The new state generated by **PERTURB** is returned to the main procedure as the current state, and its cost is assigned to the variable C_{cur}. Then the routine **UPDATE** (Figure 4.3) is invoked with p, C_{pre}, and C_{cur} as parameters. **UPDATE** compares the previous cost (C_{pre}) to the current cost (C_{cur}). If $C_{pre} = C_{cur}$, there is a good chance that the algorithm has reached a local minimum and therefore, p

procedure UPDATE$(p, Cpre, C_{cur})$;
IF $(Cpre = C_{cur})$ **THEN** /* possibility of a local minimum */
  $p = p + p_{incr}$; /* increment p to allow larger up-hill moves */
ELSE
  $p = p_0$; /* re-initialize p */
ENDIF;

Figure 4.3: The UPDATE procedure.

is increased by p_{incr} to allow larger up-hill moves.

At the end of the loop, the cost of the *current state* S is compared with the cost of the *best state* S_{best}. If S has a lower cost, then the algorithm keeps S as the best solution (S_{best}) and decrements R by ρ, thereby rewarding itself by increasing the number of iterations. This allows a more detailed investigation of the neighborhood of the newly found best solution. If S, however, has a higher cost, ρ is incremented, which is an indication of no improvements.

4.3 SELF-MAP

As mentioned earlier, the stochastic evolution algorithm [23] is a general strategy which can be used to solve a wide range of combinatorial optimization problems. The algorithm, however, has to be adapted to the type of problem under investigation. Specifically, the solution space has to be well defined and a suitable state

representation adopted, etc.

The work reported here adapts the SE algorithm to the technology mapping problem of LUT-FPGA. The developed mapper, **S**tochastic **E**volution **LUT**-**FPGA** **MAP**per (SELF-MAP), can be used to optimize either area, delay or both. The following sections detail the main features of this work.

4.3.1 The Solution Space

Background

Nodes in general Boolean networks are either fanout nodes or fanout-free nodes. Networks with only fanout-free nodes can be optimally technology mapped in linear time with algorithms like Tree-Map [6]. However, networks generally are not fanout-free, in which case, one of the following two approaches may be followed. The first approach partitions the network into a forest of fanout-free trees and a solution can be obtained by applying the tree mapping algorithm to each tree individually. The final result is obtained by re-assembling the individually mapped trees. The second approach replicates every fanout node and the whole cone that feeds it. The mapping algorithm in this case is applied to the resulting fanout-free network. Figure 4.4 illustrates the two approaches. In the figure, the original Boolean network (BN) (Figure 4.4.a) consists of one fanout node a and three cones C_1, C_2, and C_3. To

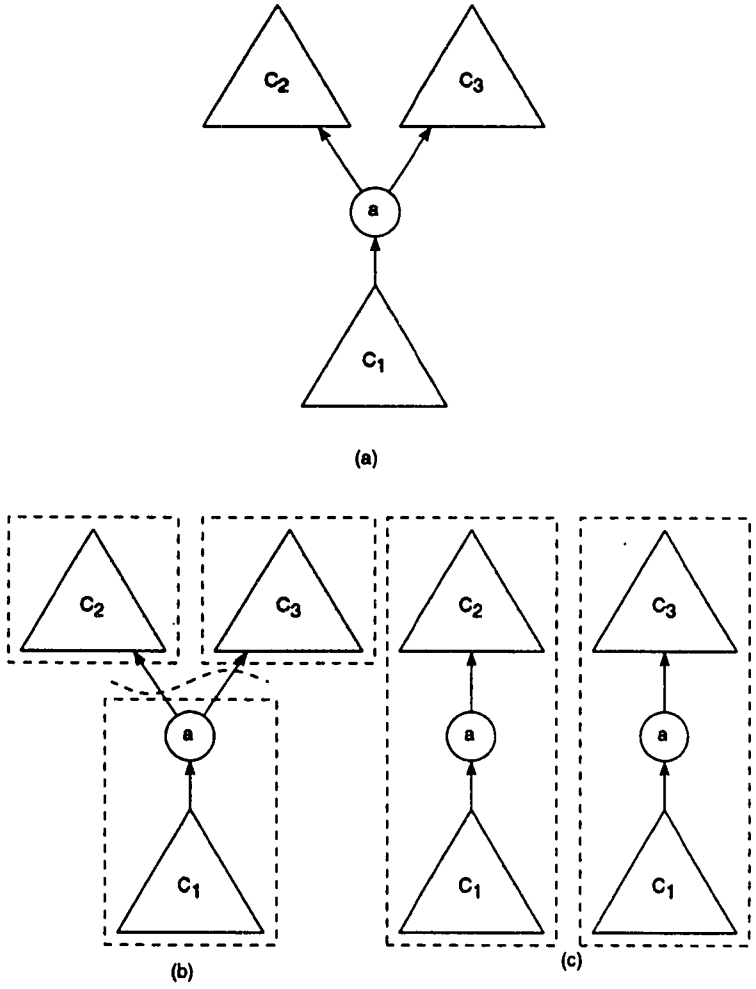


Figure 4.4: Methods of mapping networks with fanout nodes.

map BN using the first approach, BN is first partitioned by clipping the multiple fanout edges out-going from a as shown by the dotted curved line in Figure 4.4.b. As a result, BN is transformed into a forest of three trees as shown by the dotted rectangles in Figure 4.4.b. To map BN using the second approach, the fanout node a and the cone C_1 that feeds it are replicated as shown in Figure 4.4.c. As a result, BN becomes a forest of two trees as shown by the dotted rectangles in Figure 4.4.c. Following any of the above approaches, BN becomes a forest of trees. Therefore, every tree can be optimally mapped, and the final mapping is obtained by re-assembling the mapped trees.

The two approaches represent two extremes which are unlikely to yield good solutions. The search capability of the stochastic evolution algorithm [23] would suggest a more plausible approach where only part of the cone feeding a fanout node including the fanout node itself may be considered for replication. The replicated part may include some, all, or none of the nodes of the fanin cone. The resulting network is thus partitioned into a forest of trees (cones). Each tree is mapped individually and the final mapping is obtained by re-assembling the mapped networks. Obviously, the two approaches mentioned earlier are special cases of this approach. In this work, we adopted this latter approach.

Movable Objects and Locations

A mapping of a Boolean Network would assign LUTs to some of the fanout nodes, and would replicate others. A replicated fanout node implies that its immediate fanout-free predecessor nodes would turn into fanout nodes themselves and would thus be either replicated or assigned to the output of some LUT. Accordingly, the set of movable objects M is chosen to be the set of fanout and potential fanout nodes, i.e. $M = F \cup PF$, where F is the set of fanout nodes and PF is the set of potential fanout nodes.

A node $m \in M$ can either be a fanout-free node, a fanout node which has not been replicated or a fanout node which has been replicated. Accordingly, the set of locations L is chosen to be:

- Not-Replicated-Fanout (NF),
- Replicated-Fanout (RF), and
- Fanout-Free (FF).

The fanout nodes can arbitrarily move between locations, NF and RF (Figure 4.5.a). A potential fanout node (PF) should initially be in location FF. If, however, its immediate successor node is replicated, i.e. moved from location NF to location RF, the PF node is moved to location NF. From there, a potential fanout node can

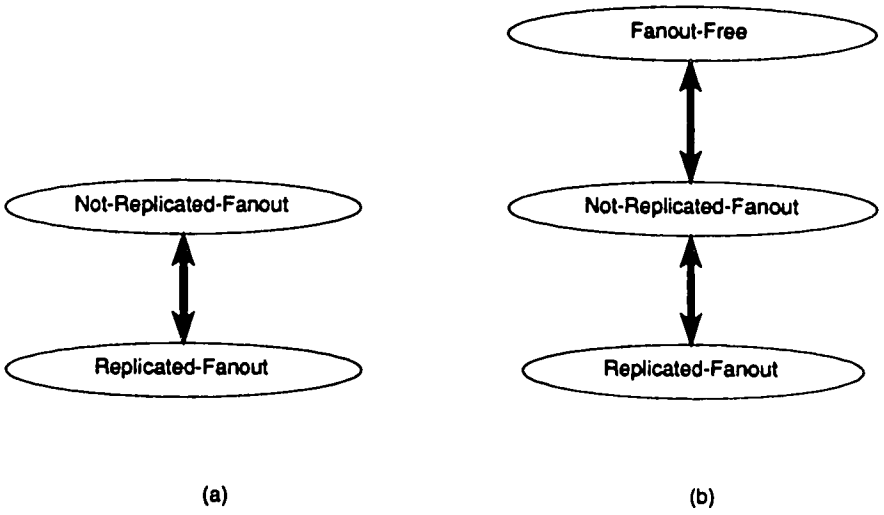


Figure 4.5: Locations of Fanout and Potential Fanout nodes.

move back and forth between locations NF and RF. If at any time, its immediate successor is un-replicated, i.e. moves back from location RF to location NF, the PF node should return back to location FF (Figure 4.5.b).

The above state model allows the SE algorithm to investigate full, partial, or no replication of all nodes in every cone that feeds any fanout node in the network. The following example illustrates these concepts.

Example: Consider the network shown in Figure 4.6. The network has four primary inputs, $PI = \{a, b, c, d\}$, two primary outputs, $PO = \{i, j\}$, two fanout nodes, $F = \{h, f\}$, and one potential fanout node, $PF = \{e\}$. Note that

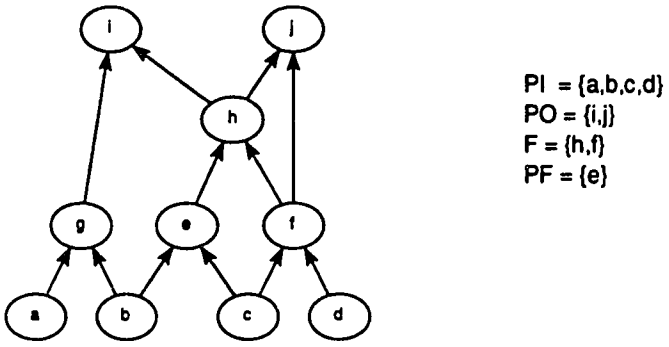
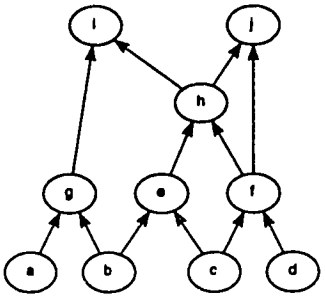


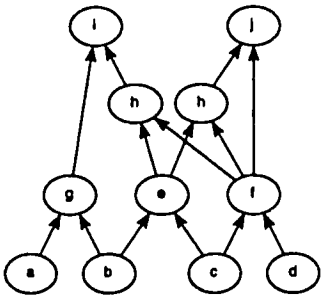
Figure 4.6: Example

primary inputs are not classified as fanout nodes, e.g. nodes b and c, or as potential fanout nodes, e.g. node d.

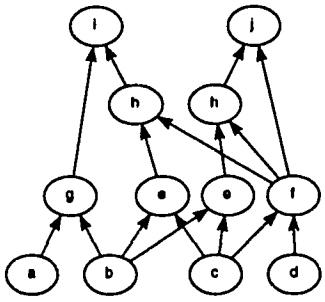
According to our state model, the set of movable objects M is chosen to be the set of fanout and potential fanout nodes. Therefore, $M = \{e, h, f\}$. Initially, h and f are in location NF, and e is in location FF. As the algorithm proceeds in searching the solution space, those movable objects change their locations. Since h and f are fanout nodes, they can only move between locations NF and RF. However, node e is a potential fanout node and thus it will be in the location FF unless h moves to location RF in which case it will be automatically moved to location NF from which it may move to location RF. If h moves back to location NF, e automatically moves to location FF. Figure 4.7 shows some possible moves and valid states in the solution space and the locations of the



FF	NF	RF
e	h, f	



FF	NF	RF
	e, f	h



FF	NF	RF
	f	e, h

Figure 4.7: Example: continue

movable objects in each case.

Ordering of the Movable Objects

It was mentioned earlier in section 4.2.2, that The **PERTURB** function scans the set of movable elements M according to some apriori ordering and moves every $m \in M$ to a new location $l \in L$. In this work, the following two orderings of movable objects have been examined.

- Random ordering, where every movable object is randomly picked and perturbed. Each object is picked only once.
- Depth First Search (DFS) ordering, where nodes closer to primary outputs are perturbed first.

To illustrate the idea, consider the previous example (Figure 4.6). The set of movable objects M is $\{e, h, f\}$. For the random ordering, the order of objects can be any one of the possible six combinations, i.e. $\{ehf, hef, hfe, feh, efh, fhe\}$. For the DFS ordering, the order of the objects can be one of two possible combinations, i.e. $\{hef, hfe\}$.

The ordering of objects seems to affect the path taken by the algorithm to reach the desired solution. However, none of these orderings is consistently better than the other. For some benchmarks, random ordering is better, and for other benchmarks,

the DFS ordering is better.

4.3.2 The Cost Function

To evaluate the fitness of a state (solution), a cost function is required. Obviously, the cost function should depend on the optimization objective. In this work, the target optimization objective is the area, delay, or both area and delay of the resulting network. The total number of LUTs in the final mapping will be used as an estimate of the required implementation area. Similarly, the depth of the final mapping will be used as a measure for the resulting circuit delay.

The cost function algorithm technology-maps an input Boolean network in a linear running time. Is is based on the algorithm reported in [6]. Upon completion, it returns a weighted sum of the number of LUTs and depth of the resulting mapping. The variables c_1 and c_2 are used to determine the weight given to area optimization as compared to delay optimization such that $c_1 + c_2 = 1$. If $c_1 = 1$ and $c_2 = 0$, the algorithm maps targeting area optimization only. If $c_1 = 0$ and $c_2 = 1$, the algorithm maps targeting delay optimization only. Other values of c_1 and c_2 provide means for obtaining other area-delay trade-off solutions.

The algorithm accepts a DAG (direct acyclic graph) representing the input Boolean network and the parameter K which is the in-degree of the LUT. It per-

ALGORITHM COST$(G(V,E), c_1, c_2)$;
  Perform a topological sort on $G(V,E)$;
  FOR every $v \in V$ **DO**
    compute d_v, z_v, and D_v ;
    IF $d_v > K$ **THEN**
      sort the immediate predecessors P of v in descending order
      of their weights, where **Weight(p)** $= c_1 \times z_p - c_2 \times D_p \times \frac{K}{D_v}$
      keep assigning LUTs to immediate predecessors of v till $d_v \leq K$;
    ENDIF
    IF v is a primary output or at location NF **THEN**
      assign a LUT to v; and
      set $z_v = 1$;
      update D_v;
    ENDIF
  ENDFOR
  RETURN($\hat{c_1}\times$ Mapping's no of LUTs + $\hat{c_2}\times$ Mapping's Depth)
END (* of COST *);

Figure 4.8: The cost algorithm.

forms a topological sort of all nodes starting from the primary inputs. For each node, the algorithm computes the node dependency d, its contribution z, and its depth D values (Figure 4.8) as follows

- Contribution z_v:

  a) $z_v = 1$ if v is a primary input or v is assigned a LUT,

  b) $z_v = \sum z_p \ \forall p$ immediate predecessor of v, otherwise.

- Dependency d_v:

  a) $d_v = 1$ if v is a primary input,

b) $d_v = \sum z_p \ \forall p$ immediate predecessor of v, otherwise.

- Depth D_v:

  a) $D_v = 0$ if v is a primary input,

  b) $D_v = max(D_p) \ \forall p$ immediate predecessor of v, if v is not assigned a LUT,

  c) $D_v = max(D_p) + 1 \ \forall p$ immediate predecessor of v, if v is assigned a LUT.

If dependency d of any node v is found to be greater than K, its immediate predecessors are sorted in a descending order according to some weight function which reflects the required relative weights assigned to area optimization versus delay optimization. LUTs are assigned to predecessor nodes with larger weights till $d_v \leq K$. Once a node is assigned to be the output of a LUT, its z value is set to 1, and the d and D values of its successor nodes are updated. The sorting weight function for a predecessor node p is chosen to be :

$$Weight(p) = c_1 \times z_p - c_2 \times D_p \times \frac{K}{D_v}$$

The choice of the *Weight* function is justified by the following argument. If the objective is area minimization, i.e. $c_1 = 1$ and $c_2 = 0$, then one should assign LUTs to nodes with larger contributions (z values) as this would more likely reduce the number of LUTs. If the objective, however, is delay minimization, i.e. $c_1 = 0$ and $c_2 = 1$, then one should assign LUTs to nodes with smaller depths (D values) as this would more likely reduce the overall depth of the mapped network. The scale factor $\frac{K}{D_v}$ limits the depth term D_p to a maximum value of K which is also the maximum

value of the contribution term z_p. Thus, the c_1 and c_2 factors would realistically reflect the required relative weight for area and/or delay optimization.

The algorithm assigns a LUT to every primary output node and every node at location NF. The z values of these nodes are set to 1 and their D values are updated.

Upon completion, the cost function returns a weighted sum of the number of LUTs and depth of the resulting mapping. This number is used by the SE algorithm to determine the fitness or goodness of a given network. Notice that the control parameters used in the weighted sum ($\hat{c_1}$ and $\hat{c_2}$) are scaled versions of c_1 and c_2. The reason is that the number of LUTs and depth of the resulting mapping are not necessarily comparable in value. Therefore, there is a need to scale the control parameters c_1 and c_2 to reflect the target relative weights for area versus delay optimization. The scaling is done as follows. The cost function is invoked only once with $c_1 = 1$ and $c_2 = 0$ (area Optimization), the number of LUTs and depth of the resulting mapping are denoted N_1 and D_1 respectively. The cost function is invoked another time but with $c_1 = 0$ and $c_2 = 1$ (delay Optimization) and the resulting number of LUTs and depth are denoted N_2 and D_2 respectively. Then, the scaled c_1 is given by $\hat{c_1} = c_1/N_{ref}$ and the scaled c_2 is given by $\hat{c_2} = c_2/D_{ref}$, where

$$N_{ref} = 0.5 \times (N_1 + N_2),$$

$$D_{ref} = 0.5 \times (D_1 + D_2).$$

The following example illustrates the way the cost function performs.

Example: Consider the Boolean network (BN) shown in Figure 4.9.a. Given that $K = 3$, $c_1 = 1$, and $c_2 = 0$. The network has four primary inputs, $PI = \{a, b, c, d\}$, two primary outputs, $PO = \{i, j\}$, one fanout node, $F = \{f\}$, and no potential fanout nodes. Accordingly, the set of movable objects $M = \{f\}$. Initially, f is in location NF since it is not replicated.

The cost algorithm starts by topologically sorting BN. Figure 4.9.b shows a possible sort. The algorithm, then, proceeds by computing the d, z and D values of every node in BN. The table at figure 4.10 shows the computed d, z and D for every node. Also, the table shows the actions taken at some nodes. For example, node f is assigned a LUT since it is at location NF (not-replicated). Accordingly, its depth D is incremented by one and its contribution z is set to one. Consider, also, node i. The dependency d of node i exceeds $K = 3$. Therefore, the immediate predecessors of node i, nodes g and h are sorted according to the *Weight* function. Node h has more weight. Therefore, it is assigned a LUT and its D value is incremented by one and its z value is set to one. Next, the d value of node i is updated. Finally, since node i is a primary output node, it is assigned a LUT and its D value is incremented by one and its z value is set to one.

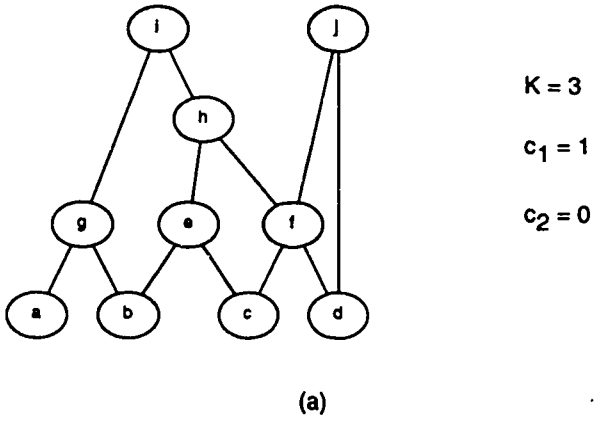


Figure 4.9: Example

node	d	z	D	actions
a	1	1	0	
b	1	1	0	
c	1	1	0	
d	1	1	0	
g	2	2	0	
e	2	2	0	
f	2	1	1	f is assigned a LUT because it is at location NF. D is updated and z is set to 1.
h	3	3	1	
i	5	1	3	Since $d_i > 3$, children of i are sorted and assigned LUTs. h is assigned a LUT.
h	3	1	2	h is assigned a LUT (see actions at i). D is updated and z is set to 1.
i	3	1	3	i is assigned a LUT because it is at location a primary output
j	3	1	3	j is assigned a LUT because it is at location a primary output

$$\text{cost} = c_1 \times \text{No of LUTs} + c_2 \times \text{Depth} = 1 \times 4 + 0 + 3 = 4$$

Figure 4.10: Example

Chapter 5

IMPLEMENTATION AND RESULTS

5.1 Introduction

We have implemented SELF-Map in the C language and tested it on a set of MCNC benchmark circuits [29]. In this chapter, we highlight some implementation aspects of this work. The experimentation results are also shown and discussed.

5.2 Input/Output Format

SELF-Map reads its input and writes its output in the *.eqn* format [27].

Table 5.1: Logic operators of Boolean expressions in .eqn format

grouping	()
XOR	!= (or ^)
XNOR	==
NOT	!
AND	& (or *)
OR	| (or +)

A typical *.eqn* file starts with the reserved words **INORDER** and **OUTORDER**. **INORDER** and **OUTORDER** are used to specify the primary inputs and primary outputs for a Boolean network respectively. These are followed by a set of equations of the form "< *signal* > = < *expr* > ;", where < *signal* > is the name of some node n, and < *expr* > is a Boolean expression which represents the functionality of this node. The Boolean expression uses the operators shown in table 5.1. Refer to the appendix for a typical *.eqn* file.

5.3 Data Structures

Two main classes of data structures have been used in implementing SELF-Map.

- NET : an adjacency list [17, 10] which contains the connectivity information of the Boolean Network,
- STATE : an adjacency list which contains the mapping information of the Boolean Network.

While SELF-Map is running, there exist one NET data structure and several STATE data structures.

5.4 Preprocessing

SELF-Map implements the technology mapping phase that has been mentioned in section 1.2. Therefore, prior to applying SELF-Map to a given Boolean network, the network undergoes a technology-independent optimization phase. The SIS [27] package has been used to perform this phase. A number of provided optimization scripts have been tried to obtain best results.

It is our belief that 2-Input Boolean networks can be more efficiently packed. Therefore, SIS is used to decompose the resulting optimized Boolean network into a 2-Input Boolean network. A 2-Input Boolean network is a Boolean network where all nodes have 2 inputs. The SIS command **xl_imp -n 2** is used to perform such decomposition.

5.5 Results

SELF-Map was tested using several MCNC benchmark circuits [29]. The circuits were first preprocessed as described in section 5.4. Mappings targeting area optimization ($c_1 = 1, c_2 = 0$) as well as mappings targeting delay optimization ($c_1 =$

Table 5.2: Results targeting area optimization

Ckt	Chortle-crf	GAFPGA	mis-pga	SELF-Map
z4ml	6	5	8	7
misex1	19	15	11	15
vg2	23	22	30	27
5xp1	28	22	31	34
count	31	31	31	39
9symml	55	57	56	52
9sym	64	56	72	42
apex7	64	61	64	67
rd84	45	43	40	25
e64	81	81	82	82
C880	86	86	103	98
alu2	112	112	129	94
duke2	123	119	128	114
C499	70	64	66	66
rot	203	200	200	214
apex6	213	194	243	200
des	955	1006	1016	908
Total	2178	2174	2310	2084
%	+4	+4	+10	1

$0, c_2 = 1$) were performed. In both cases, the results were compared to those reported in the literature. The Stochastic Evolution algorithm parameters were tuned through limited experimentation, and the following set of parameters were found to yield good results. as follows :

- $p_0 = 1$,
- $p_{incr} = 5$, and
- $R = 20$.

In case of area optimization, Table 5.2 compares the results obtained by Chortle-crf [8], GAFPGA [15], and mis-pga [18] to those obtained using SELF-Map for several MCNC benchmark circuits [29]. Out of 17 benchmark circuits, SELF-Map beats all other techniques in a total of 6 benchmarks circuits. SELF-Map results for the other benchmark circuits come somewhere in between. In total, SELF-Map provides a total saving in area of 4% over Chortle-crf, 4% over GAFPGA, and 10% over mis-pga. A striking example is the benchmark circuit *rd84*, where SELF-Map produced 44%, 42%, and 38% saving in the number of LUTs than Chortle-crf, GAFPGA, and mis-pga respectively. Another striking example is the benchmark circuit *9sym*. SELF-Map produced 34%, 25%, and 42% saving in the number of LUTs LUTs than Chortle-crf, GAFPGA, and mis-pga respectively.

In case of delay optimization, Table 5.3 compares the LUTs' depth produced by FlowMap [4], mis-pga(delay) [19], Chortle-d [9], and SELF-Map for several MCNC benchmark circuits [29]. As shown in the table, SELF-Map results are as good as the Chortle-d, slightly better than mis-pga(delay), and slightly worse than FlowMap.

5.5.1 Verification of the Results

The results obtained by SELF-Map have been verified for correctness. The SIS command **verify** [27] has been used to check for the equivalence of SELF-Map's input and output, then, the SELF-Map output has been fed to the program *count*

Table 5.3: Results targeting delay optimization

Ckt	FlowMap	mis-pga(delay)	Chortle-d	SELF-Map
z4ml	3	2	3	3
misex1	2	2	2	3
vg2	4	4	4	4
5xp1	3	2	3	3
count	3	4	4	3
9symml	5	3	5	4
9sym	5	3	5	3
apex7	4	4	4	4
rd84	4	3	4	4
C880	8	9	8	8
alu2	8	6	9	7
duke2	4	6	4	5
C499	5	8	6	6
rot	6	7	6	6
apex6	4	5	4	4
alu4	10	11	10	11
des	5	11	6	9
Total	83	90	87	87

which we have implemented to check that the the inputs to every K-LUT does not exceed K.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, a new technology mapper for Look-Up Table based Field Programmable Gate Arrays (LUT-FPGA) has been designed and implemented. The new technology mapper, SELF-Map, is based on the Stochastic Evolution Algorithm [22] and can optimize circuits for area, delay, or any area-delay combination. It is capable of mapping any arbitrary combinational Boolean network to any LUT-Based FPGA provided that the LUTs have a single output and K inputs where K is an arbitrary integer number. SELF-Map has been implemented and tested on several benchmark circuits and was shown to generally perform better than other algorithms reported

in the literature.

A brief introduction to Field Programmable Gate Arrays (FPGAs), and their typical architecture was presented in Chapter 1. A typical FPGAs CAD system and where does the technology mapping phase fit in such a system were also described therein. In Chapter 2, some related definitions were introduced, and the technology mapping problem was formally stated. Chapter 3 reviewed most of the reported technology mapping heuristic algorithms targeting LUT-based FPGAs. In Chapter 4, SELF-Map, the new technology mapper was introduced and discussed with the help of several examples. Chapter 5 highlighted some implementation aspects of this work and compared the performance of SELF-Map to six other LUT-FPGA technology mappers using several MCNC benchmark circuits. Overall, our heuristic exhibits better results compared to other reported technology mapping heuristics.

6.2 Future Work

The following items are suggested for future work.

Studying the effect of movable objects' ordering :

In this work, two ordering scheme have been investigated, namely random ordering and Depth First Search (DFS) ordering. The results show that the ordering does affect the number of iterations required to reach the desired solution. However, none of these orderings proved to be consistently better than the other. Further study of other ordering schemes may lead to a better ordering which can reduce the number of iterations taken by the Stochastic Evolution algorithm.

Studying the effect of the SE parameters :

In this work, the parameters of the Stochastic Evolution algorithm have been set as follows :

- $p_0 = 1$,
- $p_{incr} = 5$, and
- $R = 20$.

These parameters have been tuned through limited experimentations. A more thorough investigation of the effect of these parameters on the technology mapping class of problems will be another important contribution. For example, the p_{incr} or the R parameter may be dependent on the number of nodes in the input Boolean network and/or the number of movable objects.

Extending SELF-Map to map for LUTs with more than one output :

SELF-Map is capable of mapping any combinational Boolean network to any LUT-based FPGA provided that every LUT has single output and K inputs where K is an arbitrary integer number. It will be quite useful to extend SELF-Map to map for LUTs with more than one output.

Extending SELF-Map to map sequential circuits :

SELF-Map can be extended to map sequential circuits by adding some preprocessing and postprocessing routines. The input routine of SELF-Map should be extended to accept sequential circuits in some standard format. Then, there should be a routine to extract the combinational part of the input circuit. The extracted combinational part is fed to SELF-Map. The sequential part of the input circuit is then added to the output of SELF-Map.

Appendix A

This appendix shows a sample input Boolean network (Figure 6.1) to SELF-Map. The sample circuit is one of the MCNC benchmark circuits [29], namely *z4ml*. The input Boolean network is in the *.eqn* format. Figure 6.2 shows the output produced by SELF-Map for this input.

$INORDER = a\ b\ c\ d\ e\ f\ g;$
$OUTORDER = h\ i\ j\ k;$
$h = z2 + y2;$
$i = !k2 * !s2 + k2 * s2;$
$j = c3 + b3;$
$k = !g * !t2 + g * t2;$
$k2 = g3 + f3;$
$r2 = e3 + d3;$
$s2 = !b * !e + b * e;$
$t2 = !a * !d + a * d;$
$u2 = f + c;$
$v2 = c * f;$
$w2 = u2 + r2;$
$y2 = r2 * v2;$
$z2 = !j * w2;$
$b3 = c * f;$
$c3 = r2 * u2;$
$d3 = e * !i;$
$e3 = b * k2;$
$f3 = d * g;$
$g3 = a * !k;$

Figure 6.1: Input Boolean network.

$INORDER = b\ a\ g\ d\ e\ f\ c;$
$OUTORDER = k\ i\ j\ h;$
$k = !g*!(!a*!d + a*d) + g*(!a*!d + a*d);$
$k2 = (a*!k) + (d*g);$
$i = !k2*!(!b*!e + b*e) + k2*(!b*!e + b*e);$
$r2 = (b*k2) + (e*!i);$
$j = (r2*(f+c)) + (c*f);$
$z2 = !j*((f+c) + r2);$
$h = z2 + (r2*(c*f));$

$\#\ objective : 1.00*z + 0.00*depth$
$\#\ 5 - LUTs\ no : 7$
$\#\ max\ depth : 7$
$\#\ nodes : 26\ (3\ F + 4\ P + 19\ others)$
$\#\ replicated\ nodes : 1\ (1\ F + 0\ P)$
$\#\ not - replicated\ nodes : 6\ (2\ F + 4\ P)$

Figure 6.2: SELF-Map output.

Bibliography

[1] J. A. Bland and G. P. Dawson. TAbu Search and Design Optimization. *Computer Aided Design, Butterworth-Heinmann*, 23(3):195–201, April 1991.

[2] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publishers, 1992.

[3] K. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar. Dag-Map: Graph-Based FPGA Technology Mapping for Delay Optimization. *IEEE Design & Test of Computers*, pages 7–20, September 1992.

[4] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. CAD of Integrated and Systems*, 13(1):1–12, January 1994.

[5] J. Cong and Y. Ding. On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping. *IEEE Trans. on VLSI Systems*, 2(2):137–148, June 1994.

[6] Amir H. Farrahi and Majid Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Transaction of Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, November 1994.

[7] R. Francis, J. Rose, and K. Chung. Chortle: A Technology Mapping for Lookup Table-Based FPGAs. *27th DAC*, pages 613–619, 1990.

[8] R. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. *28th DAC*, pages 227–233, 1991.

[9] R. Francis, J. Rose, and Z. Vranesic. Technology Mapping of Lookup Table-Based FPGAs for Performance. *ICCAD*, pages 568–571, November 1991.

[10] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, INC., 1989.

[12] K. Karplus. Xmap: a Technology Mapper for Table-lookup Field Programmable Gate Arrays. *28th DAC*, pages 240–243, 1991.

[13] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[14] R. Kling and P. Bannerjee. ESP: A new standard cell placement package using simulated evolution . *Proceedings of 24th Design Automation Conference*, pages 60–66, 1987.

[15] V. Kommu and I. Pomeranz. GAFPGA: Genetic Algorithm for FPGA Technology Mapping. In *IEEE EURO-DAC*, pages 300–305, 1993.

[16] E. Lawler, K. Levitt, and J. Turner. Module Clustering to Minimize Delay in Digital Networks. *IEE Trans. Computers*, C-18(1):47–57, January 1969.

[17] J. A. McHugh. *Algorithmic Graph Theory.* Prentice-Hall International, Inc, 1990.

[18] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentlli. Logic Synthesis for Programmable Gate Arrays. *28th DAC*, pages 620–625, 1991.

[19] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentlli. Performance Directed Synthesis for Look Up Programmable Gate Arrays. *ICCAD*, pages 572–575, 1991.

[20] P. J. Roth and R. M. Karp. Minimization over Boolean Graphs. *IBM J. of Research and Development*, 6(2):227–238, April 1962.

[21] R. L. Rudell. Logic Synthesis for VLSI Design. *UCB/ERL Memorandum M89/49*, April 1989.

[22] Youssef G. Saab and Vasant Rao. Combinatorial Optimization by Stochastic Evolution. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 10(4):525–535, April 1991.

[23] Youssef G. Saab and Vasant B. Rao. Combinatorial Optimization by Stochastic Evolution. *IEEE Transaction of Computer-Aided Design*, 10(4):525–535, April 1991.

[24] Sadiq M. Sait and Habib Youssef. *VLSI Physical Design Automation: Theory and Practice.* McGraw-Hill Book Company, Europe, 1995.

[25] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose. Synthesis Methods for Field Programmable Gate Arrays. *Proceedings of the IEEE*, 81(7):1057–1083, July 1993.

[26] M. Schlag, J. Kong, and P. Chan. Routability-Driven Technology Mapping for Lookup Table-Based FPGA's. *IEEE Trans on CAD*, 13(1):13–26, January 1994.

[27] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. S. Vincentelli. SIS: A System for Sequential Circuit Synthesis. *Electronics Research Laboratory Memorandum*, (UCB/ERL M92/41), May 1992.

[28] Xilinx Corp. *The Programmable Gate Array Data Book*, 1994.

[29] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0.* Microelectronics Center of North Carolina, PO Box 12889, Research Triangle Park, NC 27709, January 15 1991.

Vita

- Ahmad Saleh Al-Mulhem
- Received Bachelor of Science in Electrical Engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia in August 1991.
- Worked as an Operation Engineer in Saudi Aramco from December 1991 till September 1992.
- Working as a Graduate Assistant in the Computer Engineering Department at KFUPM since September 1993.
- Completed the Master of Science in Computer Engineering from KFUPM in July 1996.